Outline

The aim is to produce an approximately compatible (at source level) but improved BASIC interpreter for the Mega65. However, I would like to produce a code base that will function on a 65C02 as well, so as to enable other systems to use it if they want.

This document provides a description of the implementation, but not of the specific features of BASIC. So initially this will run on anything at all pretty much, it would work over a Serial I/O. Features will then be added in as thought best.

The aim is to get a hopefully very robust interpreter running first.

These are going to be broadly similar to CBM Basic with the following additions as a bare minimum.

- Longer variable names.
- Named procedures/subroutines
- While and Repeat loops
- Single line IF .. THEN and multi line IF .. ELSE .. ENDIF
- Byte,Word,Long memory access.

Other likely additions are :-

- Local variables (probably not arrays)
- Procedure Parameters (ditto)
- Built in Assembler (see BBC Micro type)

Paul Robson July 2019.

Memory Usage

Both the 4510 and 65C02 operate within a 64k address space. There are requirements for the following types of storage:

- ROM Program Code
- General Data
- String Heap
- Variable/Array Definitions and Values
- BASIC Program Code
- System storage – buffers, character RAM
- Zero Page

The first and last occupy the top and bottom of memory. I estimate the interpreter size at 16k-24k depending on functionality, though some of this is likely to be banked.

General Data covers every other type of storage ; tile maps, sprites, game layouts, bitmaps, sound effects, any form of raw data. There are potentially two types of raw data – as it appears all spare memory may not be readable by VIC-IV, so it subdivides into that which can and that which can't.

String Heap and Variable/Array stuff can share a space in memory, the heap working down and variables up. Note that the string space should just be for actual strings, not for storage of general data. Given the amount of processing that is utilised here, this best belongs in the 64k processor space.

BASIC programming code is accessed using far pointers ; the (ZP),Z construct and can be anywhere, and would not normally be in the 64k processor space.

Zero page and the buffers can go at the bottom, screen memory and the like will be system dependent.

For a 65C02 system everything will go in the 64k address space, and the design needs to allow for the top of program memory to be the low of the data area.

So that gives 64k memory maps as follows (examples for Mega65 and 6502 system) :-

Memory Range	Contents
C000-FFFF	BASIC ROM (16k)
2000-C000	Variables/Heap (40k)
0000-2000	Buffers, Character RAM, etc. (8k)

Memory Range	Contents
C000-FFFF	BASIC ROM (16k)
A000-C000	General Data (8k)
Xxxx-A000	Variables/Heap (depends)
1000-xxxx	BASIC Program (depends)
0000-1000	Buffers etc.

Cross Porting Issues

Hardware

Hardware issues are largely limited to extensions designed to be supported and are very system dependent. A system should be usuable through a serial I/O port if required.

(ZP),Z Functionality

One major change to the design will be to replace the link addresses in MS Basic (which are probably a function of its origin on the 8080) with an offset, e.g. each line will be :

[offset] [line.lo] [line.hi] <tokens> [0]

Apart from no issues with relocation, this means that the code can be accessed through either (ZP),Z on the 4510 or (ZP),Y in the 65C02. Other requirements are to increment/decrement and set Y, and sometimes push/pop it. This switch can be handled via a set of macros. (The other consequence of this is limiting the line length to 255 bytes overall, but then people shouldn't write lines that long anyway)

AXYZ Functionality

Most, not all, of the 32 bit functionality will take place in the calculation code – the layer underneath the expression evaluator. This is relatively simple to handle as the AXYZ operations have equivalents on a 65C02, they just have a lot more code.

Arithmetic Unit

Because the 4510 cannot do 32 bit operations and indexing a precedence climber evaluator will have to work on a pair fixed units rather than a stack, with the fixed units being stacked/destacked as precedence changes. This suggests the use of self modifying code with 32 bit operations as the arithmetic units are going to be 8 bytes wide. (Though they will store in 5 bytes (float) 4 (integer) and 2 (string reference).

Bytes	Purpose
0-3	24 bit signed integer if integer. Mantissa with bit 31 set if floating point. 16 bit string address if string (in 0-1)
4	Mantissa. Stored here without bias, so this would be 0 for 2^0. No other meaning. This will be converted to IEE754 format when put in a variable.
5	Sign. $00 if +ve, otherwise -ve. Float only.
6	Current Precedence level
7	Type: • $00 Zero (no other values matter) • $80 Floating Point • $40 Integer • $C0 String

The Units are known as A and B, and the most common operation is A ← A op B.

Some operations will mandate integer conversion first (e.g. binary operations) some may be float only, and some will have both integer and float operations. Numbers will be converted to the correct format automatically. Unless integers are required float/integer operations will be done as floats.

Expression Evaluation

Expression evaluation will be a single precedence climber for all types, which will do type checking and coercion (e.g. float-int ops) where necessary, exposing String, Integer and Float evaluation functions and an Atomic function.

Binary Operators are:

and	or	not				Logical
>	<	>=	<=	=	<>	Comparison
&	|	^				Binary
+	-					Arithmetic
*	/	mod	>>	<<		Arithmetic/Unsigned shifts

Unary Operators are:

abs	asc	chr$	left$	mid$	right$
len	peek	peek.w	peek.l	val	str$
spc	time	not	-	&	sqr

(& is the hexadecimal marker e.g. &F7 is the same as 247)

Later additions will be the various mathematical functions:

sin	cos	tan	atn	log	exp
inkey$	inkey	get$	get		

and whatever is required to support Mega65 functions, obviously.

Program Storage and Tokenising

Programs are stored in ordered lines as normal. However, the link address is replaced by an offset, which is the first byte, 0 here indicates end of program.

Offset 1,2 are the line numbers in low high order, and the program code starts at Offset 3, and ends with a NULL token.

There are two forms of tokenising text if it is alphabetic. One is to tokenise the best match, so if the first two characters are PI then that's a PI token, not a variable. The other is to extract the identifier, possibly with a trailing (or #, and compare that against the token list.

At bottom this is an issue of : when tokenising ONTOP

is it either :-

(a) the first form Tokens [ON] [TO] and the variable P
(b) the second form A variable called ONTOP.

In many ways I like the second, but it does mean judicious spaces. If you have commands like proc *dosomething* to run a named procedure having the spaces enhances readability. However if you leave them out it thinks it's a variable called procdosomething and expects an '=' token.

Tokens range from $80-$FF. $FE-$FF are reserved as shift tokens, to be used as shifts. Token types are precedence level tokens, unary functions, commands and commands which increment and decrement the structure level which are used for automatic indentation and structure tracking. ':' is tokenised as 1, and code is stored as standard ASCII not PETSCII.

Variable Storage

xxx